There Is More Consensus in the Hierarchy

Omitted for review

Abstract

We introduce *Hierarchical Consensus* (HC), a new approach to generalizing and localizing distributed consensus. HC is well-suited for large, dynamic, and geo-replicated groups of hosts. Hierarchical Consensus uses partitions among multiple subquorums to increase overall availability, localize decision-making, and improve throughput.

The key novelty is in allowing multiple decision-making subquorums to run independently. Subquorums can be lightweight, and chosen from hosts with low mutual communication latencies. Multiple independent subquorums increase the system's overall throughput, while their small sizes and co-location allow them to reach decisions quickly. A root quorum maintains the decision-space partition and provides linearizable guarantees across the entire system. We demonstrate HC's advantages through an implementation running on Amazon EC2.

1 Introduction

Solutions to distributed consensus generally require high throughput, low latency, fault tolerance, and durability. Current approaches [6, 31, 34, 23, 39] usually assume a small number of replicas in a co-located consensus group, each centrally located on a highly available, powerful, and reliable host. These assumptions are justified by the environments in which they usually run: highly curated environments of individual data centers, centers connected by dedicated networks, or dedicated clusters.

We consider the problem of using distributed consensus to support linearizable access orderings for objects stores or global file systems. Our target environments include geo-replicated machines across the Internet with no guarantees on bandwidth, latency, or lack of partitions. We further wish to accommodate replicas with heterogeneous capabilities and usage modalities, such as systems including both highly-provisioned servers and mobile devices. This problem space is important, as it encompasses a wide variety of usages, from agglomerations of the environments assumed by previous systems down to ad hoc systems of local and personal devices.

This wider environment poses several new challenges. Widely distributed replicas might have neither high bandwidth nor low latency, and might suffer partitions of varying durations. Such systems of replicas might also be dynamic in membership, in relative location, and in workload. Straightforward approaches to running variants of Paxos [26] or Raft [36] across the wide area will perform poorly for several reasons. First, distance (in network connectivity) between consensus replicas and the most active replicas decreases the performance of the entire system. Second, network partitions in such an environment are common, either through widespread network disruption or because active devices are carried into an area of low reception. Finally, since we do not assume that the replicas running consensus are highly curated, the fault tolerance of a potentially large system can be disrupted by a very small number of unreliable hosts. Despite these issues, geo-replicating systems can be crucial for fault-tolerance, access to resources, and supporting mobility.

We propose another approach to building large systems. Rather than relying on a few replicas to provide consensus to many clients, we propose to run a consensus protocol across replicas running at or near all of those locations. The key insight of this work is that *large problem spaces can often be partitioned into mostly disjoint sets of activity without violating consistency.* We exploit this decomposition property by making our consensus protocol hierarchical, and individual consensus groups fast by ensuring they are small. We exploit locality by building subquorums from co-located replicas, and locating subquorums near clients they serve.

Our main contributions are the following:

- We describe Hierarchical Consensus, a two-tiered consensus structure that allows high throughput, localization, agility, and linearizable access to a shared namespace.
- We show how to use *delegation* to build large consensus groups that retain their fault tolerance properties while performing like small groups.
- We describe the use of *fuzzy epoch transitions* to allow global re-configurations across multiple consensus groups without forcing them into lockstep.
- We build a linearizable key-value store whose consensus group makeup and object namespace can

be rapidly re-assigned across the entire group.

- We build a sequentially-consistent replicated log from geo-replicated subquorum logs.

The challenge is in building a multi-group coordination protocol that configures and mediates the subquorums through a root quorum. The root quorum guarantees correctness by pivoting the overall system through two primary functions. First, the root quorum re-configures subquorum memberships on replica failures and system membership changes. Second, the root quorum adjusts the mapping of the object namespace to the underlying partitions. Much of the system's complexity comes from handshaking between the root quorum and the lower-level subquorums during reconfigurations.

These handshakes are made easier, and much more efficient, by using *fuzzy transitions*. Fuzzy transitions allow individual subquorums to move through re-configurations at their own pace. Given our heterogeneous, wide-area environment, forcing the entire system to transition to new configurations in lockstep would be unacceptably slow.

We validate our approach by using HC to build Alia, a linearizable [17, 2] object store explicitly intended to run with many replicas, geo-replicated across heterogeneous networks and devices. The resulting system is local, in that replicas serving clients can be located near them. The system is fast because individual operations are served by small groups of replicas, regardless of the size of the total system. The system is nimble, in that it can dynamically re-configure the number, membership and responsibilities of subquorums in response to failures, phase changes in the driving applications, or mobility among the member replicas. Finally, the system is consistent, supporting the strongest form of per-object consistency without relying on special-purpose hardware [1, 3, 4, 12, 46] or server farms.

The remainder of the paper presents the HC protocol (Section 3), describes our implementation (Section 6), evaluates HC (Section 7), discusses related work (Section 8), and concludes (Section 9).

2 Background

The canonical distributed consensus used by systems today is Paxos [25, 26]. Paxos is provably safe and designed to make progress even when a portion of the system fails. Raft [36] was designed not to improve performance, but to increase understanding of consensus behavior to better allow efficient implementations. HC uses Raft as a building block, so we describe the relevant portions of Raft at a high level, referring the reader to the original paper for complete details.

Raft: Consensus protocols typically have two phases: leader *election* and operations *commit*[1]. Raft is a strong-leader consensus protocol, which allows the election phase to be elided while a leader remains available. The protocol requires only a single communication round to commit an operation in the common case. Raft uses timeouts to trigger phase changes and provide fault tolerance. Crucially, it relies on timeouts only to provide progress, not safety. New elections occur when another replica in the quorum times out waiting for communication from the leader. Such a replica increments its *term* until it is greater than the existing leader, and announce its candidacy. Other replicas vote for the candidate if they have not seen a competing candidate with a larger term. During regular operation, clients send requests to the leader, which broadcasts `AppendEntries` messages carrying operations to all replicas. An operation is *committed* and can be executed when the leader receives acknowledgments of the `AppendEntries` message from more than half the replicas (including itself). We describe differences of our Raft implementation from canonical implementations in Section 3. Though we chose to base our protocol on Raft, a similar approach could be used to modify Paxos into a hierarchical structure.

Terms and Assumptions: Throughout the rest of this paper we use the term *root quorum* to refer to the upper, namespace-mapping and configuration-management tier of HC, and *subquorum* to describe a group of replicas (called *peers*) participating in consensus decisions for a section of the namespace. The root quorum shepherds subquorums through *epochs*, each with potentially different mappings of the namespace and replicas to subquorums. An epoch corresponds to a single commit phase of the root quorum. We use the term Raft only when describing details particular to our current use of Raft as the underlying consensus algorithm. We refer to the two phases of the base consensus protocol as the *election phase* and the *commit phase*. We use the term *vote* as a general term to describe positive responses in either phase. Epoch x is denoted e_x. Subquorum i of epoch e_x is represented as $q_{i,x}$, or just q_i when the epoch is obvious. t_a represents a specific *tag*, or disjoint subset of the namespace.

We assume faults are fail-stop [41] rather than Byzantine [28]. We do not assume that either replica hosts or networks are homogeneous, nor do we assume freedom from partitions and other network faults.

[1] Election and commit phases correspond to PROPOSE and ACCEPT phases in Paxos.

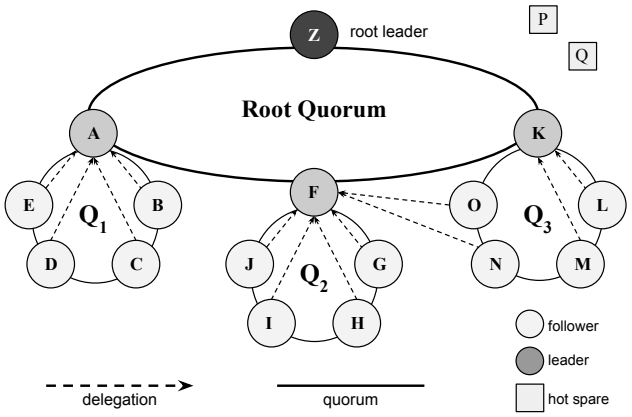


Figure 1: All replicas are members of the root quorum, but subquorum peers usually delegate their votes to their leader. Generalized delegation allows replicas r_N and r_O to delegate votes to a different leader. Hot spares r_P and r_Q are available to replace failed replicas. They participate in the root quorum even though not assigned to a subquorum.

3 Hierarchical Consensus

Hierarchical consensus is a leader-oriented protocol that organizes replicas into two tiers of quorums, each responsible for fundamentally different decisions (Figure 1). The lower tier consists of multiple independent subquorums, each committing operations to local shared logs. The upper, *root quorum*, consists of subquorum peers, usually their leaders, delegated to represent the subquorum in root elections and commits.

Hierarchical consensus's main function is to export a linearizable abstraction of shared accesses to some underlying substrate, such as a distributed object store or file system. We assume that nodes hosting object stores, applications, and HC are frequently co-located across the wide area.

Root quorums: The root quorum's primary responsibilities are mapping namespaces and replicas to individual subquorums. Each such map defines a distinct epoch, e_i, a monotonically increasing representation of the term of $q_{i,e}$. The root quorum is effectively a consensus group consisting of subquorum leaders. Somewhat like subquorums, the effective membership of the root quorum is not defined by the quorum itself, but in this case by leader election or peer delegations in the lower tier.

Subquorums: The root quorum partitions the namespace across multiple subquorums, each with a disjoint portion as its scope. The intent of subquorum localization is ensure that the *domain* of a client, the portion of the namespace it accesses, is entirely within the scope of a local, or nearby, subquorum. If true across the entire system, each client interacts with only one subquorum, and subquorums do not interact at all during execution of a single epoch. This *siloing* of client accesses simplifies implementation of strong consistency guarantees and allows better performance.

Each subquorum, q_i, elects a leader to coordinate local decisions. Fault tolerance of the subquorum is maintained in the usual way, detecting leader failures and electing new leaders from the peers. Subquorums do not, however, ever change system membership on their own. Subquorum membership is always defined in the root quorum.

Client namespace accesses are forwarded to the leader of the subquorum for the appropriate part of the namespace. The underlying Raft semantics ensure that leadership changes do not result in loss of any commits. Hence, individual- or multiple-client accesses to a single subquorum are totally ordered. *Remote accesses*, or client accesses to other than their local subquorum, are transparent to the primary protocol. However, creation of a single shared log of all system operations requires remote accesses to be logged (Section 5).

3.1 Delegation

Fault tolerance scales with increasing system size. The root quorum's membership is, at least logically, the set of all system replicas, at all times. However, running consensus elections across large systems is inefficient in the best of cases, and prohibitively slow in a geo-replicated environment. Root quorum decision-making is kept tractable by having replicas *delegate* their votes, usually to their leaders, for a finite duration. With leader delegation, the root membership effectively consists of the set of subquorum leaders. Each leader votes with a count describing its own and peer votes from its subquorum.

Consider an alternative leader-based approach where root quorum membership is defined as the current set of subquorum leaders. Both delegation and the leader approach have clear advantages in performance and flexibility over direct votes of the entire system. However, the leader approach dramatically decreases fault tolerance. Furthermore, the root quorum becomes unstable in the leader approach as its membership changes during partitions or subquorum elections. These changes would require heavyweight *joint consensus* decisions in the root quorum for correctness in Raft-like protocols [36].

With delegation, however, root quorum membership is always the entire system and remains unchanged over subquorum re-configuration. Delegation is essentially a way to optimistically shortcut contacting every replica for each decision. Subquorum repartitioning merely implies that a given replica's vote might need to be

delegated to a different leader.

Delegation does add one complication: the root quorum leader must know all vote delegations to request votes when committing epoch changes. We deal with this issue, as well as the requirement for a nuclear option (Section 3.3.2), by simplifying our protocol. Instead of sending vote requests just to subquorum leaders, **the root quorum leader sends vote requests to all system replicas.** This is true even for *hot spares*, which are not currently in any subquorum.

This is correct because vote requests now reach all replicas, and because replicas whose votes have been delegated merely ignore the request. We argue that it is also efficient, as a commit's efficiency depends only on receipt of a majority of the votes. Large consensus groups are generally slow (see Section 7) not just because of communication latency, but because large groups in a heterogeneous setting are more likely to include replicas on very slow hosts or networks. In the usual case for our protocol, the root quorum leader still only needs to wait for votes from the subquorum leaders. Leaders are generally those that respond more quickly to timeouts, so the speed of root quorum operations is unchanged.

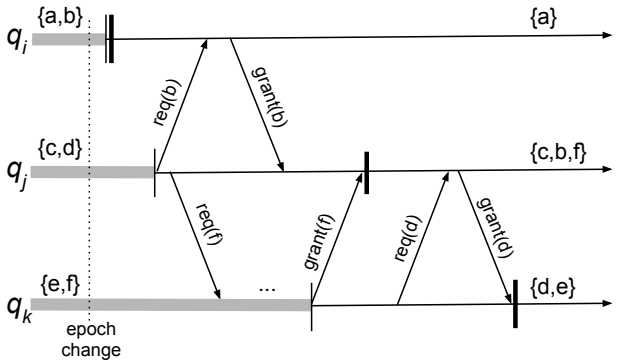


Figure 2: Readiness to transition to the new epoch is marked by a thin vertical bar; actual transition is the thick vertical bar. Thick gray lines indicate operation in the previous epoch. Subquorum q_j transitions from tag c, d to c, b, f, but begins only after receiving version information from previous owners of those tags. The request to q_k is only answered once q_k is ready to transition as well.

3.2 Epoch Transitions

An epoch change is initiated by the leader in response to one of several events, including: (i) a namespace repartition request from a subquorum leader, (ii) notification of join requests by new replicas, (iii) notification of failed replicas, and (iv) changing locations or network conditions that suggest re-assignment of replicas to existing subquorums.

The root leader transitions to a new epoch through the normal commit phase in the root quorum. The command proposed by the leader is an enumeration of the new subquorum partition, namespace partition, and assignment of namespace portions to specific subquorums. The announcement may also include initial leaders for each subquorum, with the usual rules for leader election applying otherwise, or if the assigned leader is unresponsive. Upon commit, the operation serves as an *announcement* to subquorum leaders. Subquorum leaders repeat the announcement locally, disseminating full knowledge of the new system configuration, and eventually transition to the new epoch by committing an `epoch-change` operation locally.

The epoch change is lightweight for subquorums that are not directly affected by the underlying reconfiguration. If a subquorum is being changed or dissolved, however, the *epoch-change* commitment becomes a tombstone written to the logs of all local replicas. No further operations will be committed by that version of the subgroup, and the local shared log is archived and then truncated. Truncation is necessary to guarantee a consistent view of the log within a subquorum, as peers may have been part of different subquorums, and thus have different logs, during the last epoch. Replicas then begin participating in their new subquorum instantiation. In the common case where a subquorum's membership remains unchanged across the transition, an `epoch-change` may still require additional mechanism because of changes in namespace responsibility.

Fuzzy Handshakes: Epoch handshakes are required whenever the namespace-to-subquorum mapping changes across an epoch boundary. HC separates epoch transition announcements in the root quorum from implementation in subquorums. Epoch transitions are termed *fuzzy* because subquorums need not all transition synchronously. There are many reasons why a subquorum might be slow. Communication delays and partitions might delay notification. Temporary failures might block local commits. A subquorum might also delay transitioning to allow a local burst of activity to cease such as currently running transactions[2]. Safety is guaranteed by tracking subquorum dependencies across the epoch boundary.

Example: Figure 2 shows an epoch transition where the scopes of q_i, q_j, and q_k change across the transition

[2]The HC protocol discussed in this paper does not currently support transactions.

as follows:

$$q_{i,x-1} = t_a, t_b \longrightarrow q_{i,x} = t_a$$
$$q_{j,x-1} = t_c, t_d \longrightarrow q_{j,x} = t_c, t_d, t_f$$
$$q_{k,x-1} = t_e, t_f \longrightarrow q_{k,x} = t_d, t_e$$

All three subquorums learn of the epoch change at the same time, but become ready with varying delays. These delays could be because of network lags or ongoing local activity. Subquorum q_i gains no new tags across the transition and moves immediately to the new epoch. Subquorum q_j's readiness is slower, but then it sends requests to the owners of both the new tags it acquires in the new epoch. Though q_i responds immediately, q_k delays its response until locally operations conclude. Once both handshakes are received, q_j moves into the new epoch, and q_k later follows suit.

These bilateral handshakes allow an epoch change to be implemented incrementally, eliminating the need for lockstep synchronization across the entire system. This flexibility is key to coping with partitions and varying connectivity in the wide area. However, this piecewise transition, in combination with subquorum re-definition and configuration at epoch changes, also means that individual replicas *may be part of multiple subquorums at a time*.

This overlap is possible because replicas may be mapped to distinct subgroups from one epoch to the next. Consider q_k in Figure 2 again. Assume the epochs shown are e_x and e_{x+1}. A single replica, r_a, may be remapped from subquorum $q_{k,x}$ to subquorum $q_{i,x+1}$ across the transition. Subquorum $q_{k,x}$ is late to transition, but $q_{i,x+1}$ begins the new epoch almost immediately. Requiring r_a to participate in a single subquorum at a time would potentially delay $q_{i,x+1}$'s transition and impose artificial synchronicity constraints on the system. One of the many changes we made in the base Raft protocol is to allow a replica to have multiple distinct shared logs. Smaller changes concern the mapping of requests and responses to the appropriate consensus group.

3.3 Fault Tolerance

We assert that consensus at the leaf replicas is correct and safe because decisions are implemented using well-known leader-oriented consensus approaches. Hierarchical consensus therefore has to demonstrate linearizable correctness and safety between subquorums for a single epoch and between epochs. Briefly, linearizability requires external observers to view operations to objects as instantaneous events. Within an epoch, subquorum leaders serially order local accesses, thereby guaranteeing linearizability for all replicas in that quorum.

Epoch transitions raise the possibility of portions of the namespace being re-assigned from one subquorum to another, with each subquorum making the transition independently. Correctness is guaranteed by an invariant requiring subquorums to delay serving newly acquired portions of the namespace until after completing all appropriate handshakes.

3.3.1 Failures

During failure-free execution, the root quorum partitions the system into disjoint subquorums, assigns *subquorum leaders*, and assigns partitions of the tagspace to subquorums. Each subquorum coordinates and responds to accesses for objects in its assigned tagspace. We define the system's *safety* property as guaranteeing that non-linearizable (or non-sequentially-consistent, see Section 5) event orderings can never be observed. We define the system's *progress* property as the system having enough live replicas to commit votes or operations in the root quorum.

The system can suffer several types of failures, as shown in Table 1. Failures of subquorum and root quorum leaders are handled through the normal consensus mechanisms. Failures of subquorum peers are handled by the local leader petitioning the root quorum to reconfigure the subquorum in the next epoch. Failure of a root quorum peer is the failure of subquorum leader, which is handled as above. Root quorum heartbeats help inform other replicas of leadership changes, potentially necessary when individual subquorums break down.

HC's structure means that some faults are more important than others. Proper operation of the root quorum requires the majority of replicas in the majority of subquorums to be non-faulty. Given a system with $2m + 1$ subquorums, each of $2n + 1$ replicas, the entire system's progress can be halted with as few as $(m+1)(n+1)$ well-chosen failures. Therefore, in worst case, the system can only tolerate:

$$f_{worst} = mn + m + n$$

failures and still make progress. At maximum, HC's basic protocol can tolerate up to:

$$f_{best} = (m + 1) * n + m * (2n + 1) = 3mn + m + n$$

failures. As an example, a 25/5 system can tolerate at least 8 and up to 16 failures out of 25 total replicas. A 21/3 system can tolerate at least 7, and a maximum of 12, failures out of 21 total replicas. Individual subquorums might still be able to perform local operations despite an impasse at the global level.

Total subquorum failure can temporarily cause a portion of the namespace to be unserved. However, the root quorum eventually times out and moves into

Failure Type	Response
subquorum peer	request replica repartition from root quorum
subquorum leader	local election, request replacement from root quorum
root leader	root election (with delegations)
majority of majority of subquorums	(nuclear option) root election after delegations timed out

Table 1: Failure categories: Peer failure is detected by missed heartbeat messages. The rest are triggered by the appropriate election timeout.

a new epoch with that portion assigned to another subquorum.

3.3.2 The Nuclear Option

Singleton consensus protocols, including Raft, can tolerate just under half of the entire system failing. As described above, HC's structure makes it more vulnerable to clustered failures. Therefore we define a *nuclear option*, which uses direct consensus decision among all system replicas to tolerate any f replicas failing out of $2f + 1$ total replicas in the system.

A nuclear vote is triggered by the failure of a root leader election. A *nuclear candidate* increment's its term for the root quorum and broadcasts a request for votes to all system replicas. The key difficulty is in preventing delegated votes and nuclear votes from reaching conflicting decisions. Such situations might occur when temporarily unavailable subquorum leaders regain connectivity and allow a wedged root quorum to unblock. Meanwhile, a nuclear vote might be concurrently underway.

Replica delegations are defined as intervals over specific slots. Using local subquorum slots would fall prey to the above problem, so we define delegations as a small number (often one) of root slots, which usually correspond to distinct epochs. During failure-free operation, peers delegate to their leaders and are all represented in the next root election or commit. Peers then renew their delegations to their leaders by appending them to the next local commit reply. This approach works for replicas that change subquorums over an epoch boundary, and even allows peers to delegate their votes to arbitrary other peers in the system (see replicas r_N and r_O in Figure 1).

This approach is simple and correct, but deals poorly with leader turnovers in the subquorums. Consider a subquorum where all peers have delegated votes to their leader for the next root slot. If that leader fails, none of the peers will be represented. We finesse this issue by re-defining such delegations to count root elections, root commits, *and* root heartbeats. The latter means that local peers will regain their votes for the next root quorum action if it happens after to the next heartbeat.

Example: Consider the worst-case failure situation discussed in Section 3.3.1: a majority of the majority of subquorums have failed. None of the failed subquorum leaders can be replaced, as none of those subquorums have enough local peers.

The first response is initiated when a replica holding delegations (or its own vote) times out waiting for the root heartbeat. That replica increments its own root term, adopts the prior system configuration as its own, and becomes a root candidate. This candidacy fails, as a majority of subquorum leaders, with all of their delegated votes, are gone. Progress is not made until delegations time out. In our default case where a delegation is for a single root event, this happens after the first root election failure.

At the next timeout, any replica might become a candidate because delegations have lapsed (under our default assumptions above). Such a *nuclear* candidate increments its root term and sends candidate requests to all system replicas, succeeding if it gathers a majority across all live replicas.

The first candidacy assumed the prior system configuration in its candidacy announcement. This configuration is no longer appropriate unless some of the "failed" replicas quickly regain connectivity. Before the replica announces its candidacy for a second time, however, many of the replica replies have timed out. The candidate alters its second proposed configuration by recasting all such replicas as hot spares and potentially reducing the number and size of the subgroups. Subsequent epoch changes might re-integrate the new hot spares if the replicas regain connectivity.

4 The Alia Key-Value Store

Alia is a linearizable key-value store implemented using Hierarchical Consensus. Replica structure is shown in Figure 3. Alia maps onto HC by using the namespace to represent the object space, and subquorum operations to commit object writes. Reads are not committed by default, but are always served by the leader of the appropriate subquorum. Namespace assignments in HC result in the object space being partitioned (or sharded) across distinct subquorums.

The mapping of the shared object namespace to individual subquorums is the *tagset*, or the tagset par-

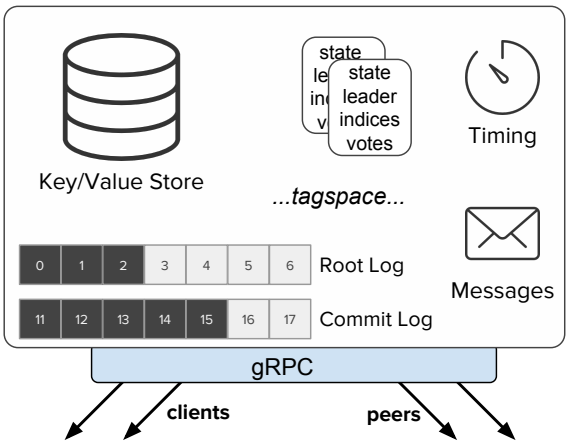


Figure 3: Alia replica. State includes both root quorum and subquorum

tition. An individual *tag* defines a disjoint subset of the object space. Subquorum membership, subquorum leaders, and tagsets are all proposed and committed in epoch changes in the root quorum.

As Alia commits writes through HC, the shared logs provide a complete version history of all distributed objects. Subquorum leaders use in-core caches to provide fast access to recently accessed objects in the local subquorum's tag. Replicas perform background anti-entropy [13, 42, 38], disseminating log updates a user-defined number of times across the system.

The most complex portion of the Alia protocol is in handling data-related issues at epoch transitions. Transitions may cause tags to be transferred from one subquorum to another, forcing the new leader to load state remotely to serve object requests. Transitions handshakes are augmented in three ways. First, an Alia replica can demand-fetch an object version from any other system replica. Second, epoch handoffs contain enumerations of all current object versions, though not the data itself. Knowing an object's current version gives the new handler of a tag the ability to demand fetch an object that is not yet present locally. Finally, handshakes start immediate fetches of the in-core version cache from the leader of the tag's subquorum in the old epoch to the leader in the new.

We do not currently gather the entire shared log onto a single replica because of capacity and flexibility issues. Capacity is limited because our system and applications are expected to be long-lived. Flexibility is a problem because HC, and applications built on HC, gain much of their value from the ability to pivot quickly, whether to deal with changes in the environment or for changing application access patterns. We require handoffs to be as lightweight as possible to preserve this advantage.

Pushing all writes through subquorum commits and serving reads at leaders allows us to guarantee that accesses are linearizable (Lin), which is the strongest non-transactional consistency [17, 2]. As a recap, linearizability is a combination of atomicity and timeliness guarantees about accesses to a single object. Both `reads` and `writes` must appear atomic, and also instantaneous at some time between a request and the corresponding response to a client. `Reads` must always return the latest value. This implies that reads return values are consistent *with any observed ordering*, i.e., the ordering is *externalizable* [15].

Linearizability of object accesses can be *composed*. If operations on each object are linearizable, the entire object space is also linearizable. This allows our subquorums to operate independently while providing a globally consistent abstraction.

5 Globally Consistent Logs

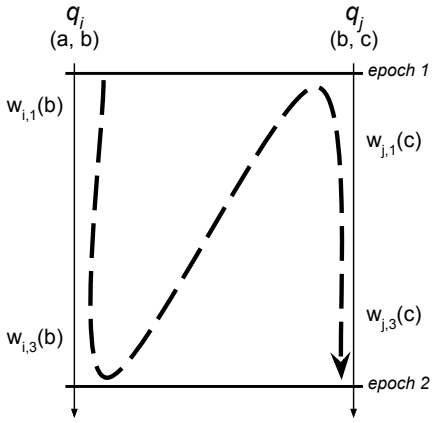
Our default use case is in providing linearizable access to an object store. Though this approach allows us to guarantee all observers will see linearizable results of object accesses in real-time, the system is not able to enumerate a total order, or create a linearizable shared log. Such a linear order would require fine-grained (expensive) coordination across the entire system, or fine-grained clock synchronization [12]. Though many or most distributed applications (objects stores, file systems, etc.) will work directly with HC, shared logs are a useful building block for distributed systems.

HC *can* be used to build a sequentially consistent (SC) shared log. Like Lin, SC requires all observers to see a single total ordering. SC differs in that this total ordering does not have to be externalizable. Instead, it merely has to conform to local operation orders and all reads-from dependencies.

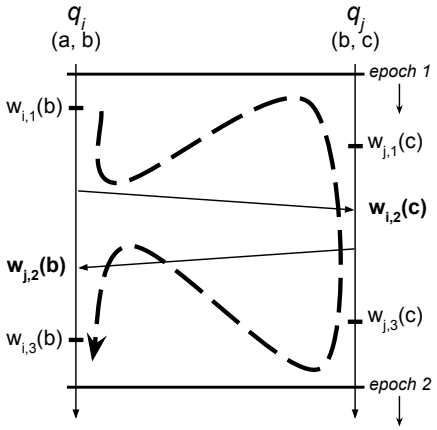
Figure 4(a) shows a system with subquorums q_i and q_j, each of which performs a pair of writes. Without cross-subquorum reads or writes, ordering either subquorum's operations first creates a SC total ordering: $q_i \rightarrow q_j$ ("happened-before" [24]) implies $w_{i,1} \rightarrow w_{i,3} \rightarrow w_{j,1} \rightarrow w_{j,3}$, for example.

By contrast, the subquorums in Figure 4(b) create additional dependencies by issuing remote writes to other subquorums: $w_{i,2} \rightarrow w_{j,3}$ and $w_{j,2} \rightarrow w_{i,3}$. Similar dependencies result from remote reads.

These dependencies cause the epochs to be split (not shown in picture). The receipt of write $w_{i,2}$ in q_j causes $q_{j,1}$ to be split into $q_{j,1.1}$ and $q_{j,1.2}$. Likewise, the receipt of write $w_{j,2}$ into q_i causes q_i to be split into $q_{i,1.1}$ and $q_{i,1.2}$. Any topological sort of the subepochs that respects these orderings, such as $q_{i,1.1} \rightarrow q_{j,1.1} \rightarrow q_{j,1.2} \rightarrow q_{i,1.2}$, results in a valid SC ordering.



(a) Default ordering: $w_{i,1} \rightarrow w_{i,3} \rightarrow w_{j,1} \rightarrow w_{j,3}$



(b) Remote writes add additional ordering constraints: $w_{i,1} \rightarrow w_{i,2} \rightarrow w_{j,3}$, and $w_{j,1} \rightarrow w_{j,2} \rightarrow w_{i,3}$

Figure 4: Dotted lines show one possible event ordering for replicas G_1 (responsible for objects a and b), and G_2 (c and d). Each remote write establishes a partial ordering between events of the sender before the sending of the write, and the writes by the receiver after the write is received.

Presenting a sequentially consistent global log across the entire system, then, only requires tracking these inter-subquorum data accesses, and then performing an $\mathcal{O}(n)$ merge of the subepochs.

Versus linearizability: By definition, this log's ordering respects any externally visible ordering of cross-subquorum accesses (accesses visible to the system). However, the log does not necessarily order other accesses according to external visibility. The resulting shared log could not be mined to find causal relationships between accesses through external communication paths unknown to the system.

For example, assume that log events are published posts, and that one user claimed plagiarism. The accused would not be able to prove that his post came first unless there were some causal chain of posts and references visible to the protocol.

6 Implementation

Alia implements a replicated key-value store with strong consistency via the Hierarchical Consensus protocol. An Alia cluster is composed of N replica processes that each manage a partial replica of the object space. Hierarchical Consensus with modified Raft as the underlying consensus protocol exports a linearizable order of accesses to the distributed key-value store. Alia and the HC library are implemented in Golang use gRPC [40] for communication. The system is implemented in 7,924 lines of code, not including standard libraries or support packages.

Each Alia replica (Figure 3) implements an event loop that responds to timing events, client requests, and messages from peers. Events may cause the replica to change state, modify a command log, broadcast messages to peers, modify the key-value store, or respond to a client. Event handlers need to aggressively lock shared state for correctness because Golang and gRPC make extensive use of multi-threading. The balance between correctness and concurrency-driven performance leads to increasing complexity and tighter coupling between components, one that foreshadows extra-process consistency concerns that have been noted in other work [9, 36].

Consistency: Alia guarantees completely linearizable data accesses to individual objects. An object `write` must be committed to the subquorum log before it can be applied to the key-value store. Any `write` that is not successfully committed is dropped without being applied to the store, requiring clients to retry the `write`. Object `reads` are serialized with respect to all `writes` by ensuring they are satisfied by the leader. Non-leader replicas redirect clients to the appropriate leader for the requested object. Alia also supports a more relaxed model where `reads` can be satisfied by any replica in the subquorum from values in the shared log [44]. Shared logs on peer replicas are not guaranteed to be up-to-date, though the degree of staleness can be parameterized.

Timing: The computing and network environment of a distributed system plays a large role in determining not just the performance of the system, but also its behavior. A simple example is the election timeout parameter of the Raft consensus protocol, which must be

Name	Time	Actions
sub heartbeat	1T	sub leader heartbeat
sub leader	2-4T	new sub election
root heartbeat	10T	root leader heartbeat
root election	20-40T	new root election
obligation	50T	root quorum may reallocate the tag

Table 2: Parameterized timeouts in our implementation. The *obligation* timeout stops a partitioned subquorum after an extended time without contact to the rest of the system. $T = 10$ msec for our experiments on Amazon EC2.

much greater than the average time to broadcast and receive responses, and much less than the mean time between failures [36, 32]. If this requirement is not met, leader may be displaced before heartbeat messages arrive, or the system will be unable to recover when a leader fails. As a result, the relationship between timeouts is critically dependent on the mean latency (λ_μ) of the network. Howard [19] proposes $T = \lambda_\mu + 2\lambda_\sigma$ to determine timeouts based on the distribution of observed latencies, sets the heartbeat as $\frac{T}{2}$, and the election timeout as the interval $U(T, 2T)$. We parameterize our timeouts (Table 2) on latency measurements made before we ran our experiments. Monitoring and adapting to network conditions is part of ongoing work.

Protocol: An Alia replica implements multiple instantiations of the Raft protocol, which we have modified in several ways. Every replica must run one instantiation of the *root consensus protocol*. Replicas may also run one or more instantiations of the *commit consensus protocol* if they are assigned to a subquorum. Vote delegation is the subject of Section 3.1. Repartition decisions move the system between epochs with a new configuration and tagspace, and can only be initiated by messages from peers or monitoring processes. A successful repartition results in a new epoch, tagspace, and subquorum topology committed to the root log. `Repartition` messages also serve to notify the network about events that do not require an epoch change, such as the election of a new subquorum leader or bringing a failed node back online.

Changes to base Raft: In addition to major changes, such allowing replicas to be part of multiple quorums simultaneously, we also made many smaller changes that had pervasive effects. One change was including the *epoch* number alongside the term in all log entries. The epoch is evaluated for invariants such as whether or not a replica can append an entry or if a log is as up to date as a remote log.

Vote delegation requires changes to vote counting. Since our root quorum membership actually consists of the entire system, all replicas are messaged during root events. All replicas reply, though most with a "zero votes" acknowledgment. The root uses observed vote distributions to inform the ordering of future consensus messages (sending requests first to replicas with votes to cast), and uses timeouts to move non-responsive replicas into "hot spares" status.

We allow `AppendEntries` requests in subquorums to aggregate multiple client requests into a single consensus round. Such requests are collected while an outstanding commit round is ongoing, then sent together when that round completes. The root quorum also aggregates all requests within a minimum interval into a single new epoch-change/reconfiguration operation to minimize disruption.

Commits are observed by the leader once a majority of replicas respond positively. Other replicas learn about the commit only on the next message or heartbeat. Root epoch changes and heartbeats are designed to be rare, meaning that epoch change commits are not seen promptly. We modified the root protocol to inform subquorums of the change by sending an additional heartbeat immediately after it observes a commit.

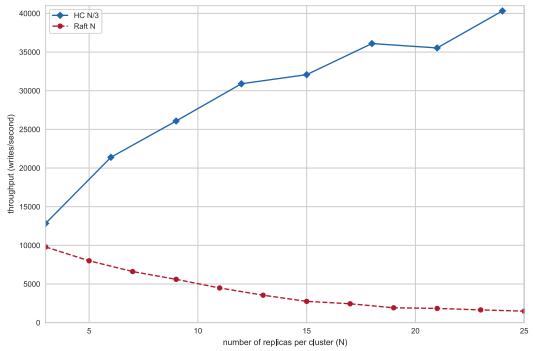
Replicas may be part of both a subquorum and the root quorum, and across epoch boundaries may be part of multiple subquorums. In principle, a high performance replica may participate in any number of subquorums. We therefore allow replicas to accommodate multiple distinct logs with different access characteristics.

Peers that are either slow or with unsteady connectivity are occasionally left behind at subquorum leader or epoch changes. Root heartbeats containing the current system configuration are broadcast to all replicas and serve to bring them up to date.

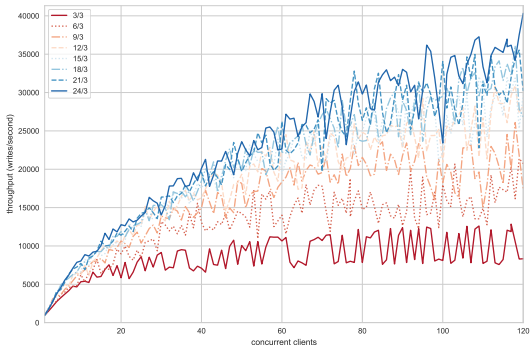
Finally, consensus protocols often synchronously write state to disk before responding to remote requests. This allows replicas that merely crash to reboot and rejoin the ongoing computation after recovering state from disk. Otherwise, these replicas need to go through heavyweight leave-and-rejoin handshakes. Our system avoids these synchronous writes by allowing epochs to re-join a subquorum at the next epoch change without any saved state, avoiding these handshakes altogether.

7 Evaluation

Alia was designed to adapt both to dynamic workloads as well as variable network conditions. We therefore evaluate Alia in two distinct environments: a homogeneous data center and a heterogeneous real-world network. The homogeneous cluster is hosted on Amazon EC2 and includes 26 "t2.medium" instances:



(a) Mean throughput of workloads of up to 120 clients.



(b) Throughput vs workload for different HC cluster sizes.

Figure 5: Performance of distributed consensus with an increasing workload of concurrent clients. Performance is measured by throughput, the number of writes committed per second.

dual-core virtual machines running in a single VPC with inter-machine latencies of $\lambda_\mu = 0.399ms$ and $\lambda_\sigma = 0.216ms$. These machines are cost effective and, though lightweight, are easy to scale to large cluster sizes as workload increases. Experiments are set up such that each instance runs a single replica process and multiple client processes.

The heterogeneous cluster (UMD) consists of several local machines distributed across a wide area, with inter-machine latencies ranging from $\lambda_\mu = 2.527ms$, $\lambda_\sigma = 1.147ms$ to $\lambda_\mu = 34.651ms$, $\lambda_\sigma = 37.915ms$. Machines in this network are a variety of dual and quad core desktop servers that are solely dedicated to running these benchmarks. Experiments on these machines are set up so that each instance runs multiple replica and client processes co-located on the same host. In this environment, localization is critical both for performance but also to ensure that the protocol can elect and maintain consensus leadership. The variability of this network also poses challenges that Alia is uniquely suited to handle via root quorum-guided adaptation. We explore two distinct scenarios in Figure 7 using this cluster; all other experiments were run on the EC2 cluster.

Scaling: Alia is partially motivated by the need to scale strong consistency to large cluster sizes. We based our work on the assumption that consensus performance decreases as the quorum size increases, which we confirm empirically in Figure 5(a). This figure shows the maximum throughput against system size for a variety of workloads, up to 120 concurrent clients. A workload consists of one or more clients continuously sending writes of a specific object or objects to the cluster without pause.

Standard consensus algorithms, Raft in particular, scale poorly with uniformly decreasing throughput as nodes are added to the cluster. Commit latency increases with quorum size as the system has to wait for more responses from peers, thereby decreasing overall throughput. Figure 5 clearly shows the multiplicative advantage of Alia's hierarchical structure, though Alia does not scale linearly as we had expected.

There are at least two factors currently limiting the HC throughput shown here. First, the HC subquorums for the larger system sizes are not saturated. A single 3-node subquorum saturates at around 25 clients and this experiment has only about 15 clients per subquorum for the largest cluster size. We ran experiments with 600 clients, saturating all subquorums even in the 24-node case. This throughput peaked at slightly over 50,000 committed writes per second, better but still lower than the linear scaling we had expected.

We think the reason for this ceiling is hinted at by Figure 5(b). This figure shows increasingly larger variability with increasing system sizes. A more thorough examination of the data shows widely varying performance across individual subquorums in the larger configurations. We suspect that the cause is either VM misconfiguration or misbehavior. We are adding more instrumentation to diagnose the problem.

The effect of saturation is also demonstrated in Figure 6, which shows cumulative latency distributions for different system sizes holding the workload (number of concurrent clients) constant. The fastest (24/3) shows nearly 80% of client write requests being serviced in under 2 msec. Larger system sizes are faster because the smaller systems suffer from contention (25 clients can saturate a single subquorum). Because throughput is directly related to commit latency, throughput variability can be mitigated by adding additional subquorums to balance load.

Adaptivity: Besides pure performance and scaling,

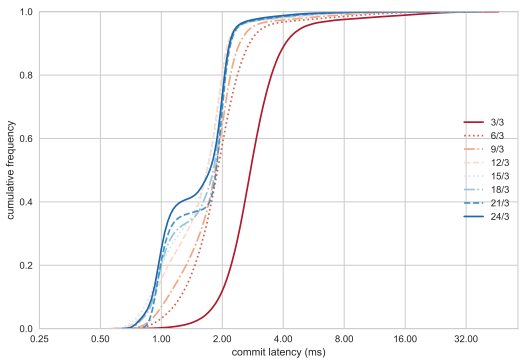


Figure 6: Client-observed cumulative latency distributions with 25 clients versus system size.

Alia is also motivated by the need to adapt to varying environmental conditions. In the next set of experiments, we explore two common runtime scenarios that motivate adaptation: shifting client workloads and failures. We show that Alia is able to adapt and recover with little loss in performance. These scenarios are shown in Figure 7 as throughput over time, where vertical dotted lines indicate an epoch change.

The first scenario, described by the time series in Figure 7(a) shows an Alia 3-replica configuration moving through two epoch changes. Each epoch change is triggered by the need to localize tags accessed by clients to nearby subquorums. The scenario shown starts with all clients co-located with the subquorum serving the tag they are accessing. However, clients incrementally change their access patterns first to a tag located on one remote subquorum, and then to the tag owned by the other. In both cases, the root quorum adapts the system by repartitioning the tagspace such that the tag defining their current focus is served by the co-located subquorum.

Finally, Figure 7(b) shows a 3-subquorum configuration where one entire subquorum becomes partitioned from the others. After a timeout, the root uses an epoch change to re-allocate the tag of the partitioned subquorum over the two remaining subquorums. The partitioned subquorum eventually has an *obligation timeout*, after which the root quorum is not obliged to leave the tag with the current subquorum. The tag may then be re-assigned to any other subquorum. Timeouts are structured such that by the time an obligation timeout fires, the root quorum has already re-mapped that subquorum's tag to other subquorums. As a result, the system is able to recover from the partition as fast as possible. Note that in this figure, the repartition occurs through two epoch changes, the first allocating part of the tagspace to the first subquorum, and the second allocating the rest of the tag to the other. Gaps in the graph are periods where the subquorums are electing local leaders. This may be optimized by having leadership assigned or maintained through root consensus.

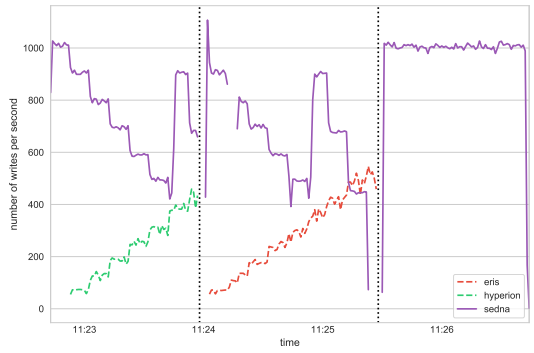
8 Related Work

Our principle contribution is Hierarchical Consensus, a general technique to compose consensus groups, maintain consistency invariants over large systems, and adapt to changing conditions and application loads. HC is related to the large body of work improving throughput in distributed consensus over the Paxos protocol [25, 33, 18, 29], and on Raft [36, 20]. These approaches focus on fast vs. slow path consensus, eliding phases with dependency resolution, and load balancing.

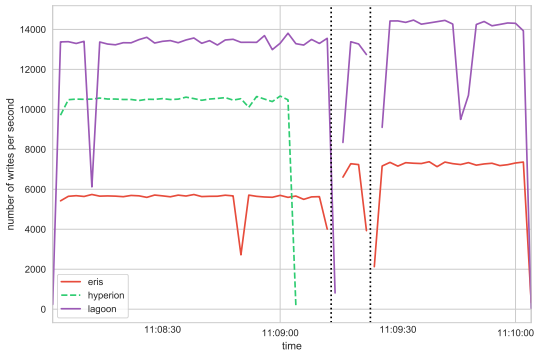
Our work is also orthogonal in that subquorums and the root quorum can be implemented with different underlying protocols, though the two levels must be integrated quite tightly. Further, HC abstracts reconfiguration away from subquorum consensus, allowing multiple subquorums to move into new configurations and reducing the need for joint consensus [36] and other heavyweight procedures. Finally, its hierarchical nature allows the system to multiplex multiple consensus instances on disjoint partitions of the object space while still maintaining global consistency guarantees.

The global consistency guarantees of HC are in direct contrast to other systems that scale by exploiting multiple consensus instances [10, 23, 12] on a per-object basis. These systems retain the advantage of small quorum sizes but cannot provide system-wide consistency invariants. Another set of systems uses quorum-based decision-making but relaxes consistency guarantees [13, 11, 30]; others provide no way to pivot the entire system to a new configuration [16]. Chain replication [45] and Vertical Paxos [27] are among approaches that control Paxos instances through other consensus decisions. However, HC differs in the deep integration of the two different levels. Whereas these approaches are top down, HC consensus decisions at the root level replace system configuration at the subquorum level, and vice versa.

Possibly the closest system to HC is Scatter [16], which uses an overlay to organize consistent groups into a ring. Neighbors can join, split, and talk amongst themselves. The bottom-up approach potentially allows scaling to many subquorums, but the lack of central control makes it hard to implement global re-maps beyond the reach of local neighbors. HC ties root quorum and subquorums tightly together, allowing root quorum decisions to completely reconfigure the running system on the fly either on demand or by detecting



(a) 9/3 system adapting to changing client access patterns by repartitioning the tag space so that clients are co-located with subquorums that serve tags they need.



(b) 9/3 System that adapts to failure (partition) of entire subquorum. After timeout, the root quorum re-partitions the tag allocated to the failed subquorum among the other two subquorums.

Figure 7: Both experiments with systems of 9 replicas arranged into 3 subquorums.

changes in network conditions.

We claim very strong consistency across a large distributed system, similar to Spanner [12]. Spanner provides linearizable transactions through use of special hardware and environments, which are used to tightly synchronize clocks in the distributed setting. Spanner therefore relies on a very specific, curated environment. HC targets a wider range of systems that require cost effective scaling in the data center to rich dynamic environments with heterogeneity on all levels.

Finally, shared logs have proven useful in a number of settings from fault tolerance to correctness guarantees. However, keeping such logs consistent in even a single consensus instance has proven difficult [8, 14, 21]. More recent systems are leveraging hardware support to provide fast access to shared logs [46, 4, 44, 43, 5, 1]. To our knowledge, HC is the first work to propose synchronizing shared logs across multiple discrete consensus instances in the wide area.

9 Conclusions and Discussion

This paper has shown the design and evaluation of Hierarchical Consensus in the context of the Alia key-value store. The system performance and agility rests on the ability to partition the namespace among a number of small, fast subquorums. The keys to making the system as a whole fast are flexible couplings between the various levels. The first technique is generalized delegation, which allows global votes to be decided by small quorums in the best case, and by the entire system's membership otherwise. Delegation allows most global decisions to be fast, while preserving the fault tolerance of a much larger quorum. Fuzzy epoch transitions allow subquorums to transition to new epochs at different times, minimizing the waits that would be required if they transitioned in lockstep. Hierarchical consensus's performance results from running subquorums in parallel. Its agility rests in the ability of a small set of replicas to make global decisions, and to pivot the entire system into new configurations as a result.

Remapping and repartitioning decisions are currently informed by simple heuristics. Our future work includes equipping the system with online monitoring of local network conditions and access patterns. We propose that decentralized administration through machine learning techniques will allow the system to be as responsive as possible, involving users in *active learning* [22, 37] while taking advantage of immediate local optimizations [35]. The application of supervised and unsupervised models will allow the system to use historical data to automatically make administrative decisions for a wide space of instances.

An example of configuration optimization through online bandit algorithms [7] involves the anti-entropy of shared logs across the system. Timely propagation of logs to all nodes in a large network depends on the random selection of a neighbor with which to perform anti-entropy. Instead of simply implementing uniform random selection, a multi-armed bandit can be used to select "better" neighbors as gossip partners. Reinforcement should prefer neighbors that have not seen updates and whose latency is as low as possible. In this way, an anti-entropy topology can be learned, and updated, such that propagation is fast.

Complete source code and documentation will be available at `http://github.com/.../alia` by the time of the camera-ready version.

References

[1] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 1–14.

[2] Attiya, H., and Welch, J. L. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS) 12*, 2 (1994), 91–122.

[3] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. Corfu: A shared log design for flash clusters. In *NSDI* (2012), pp. 1–14.

[4] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 325–340.

[5] Bernstein, P. A., Reid, C. W., and Das, S. Hyder-a transactional record manager for shared flash. In *CIDR* (2011), vol. 11, pp. 9–12.

[6] Biely, M., Milosevic, Z., Santos, N., and Schiper, A. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on* (2012), IEEE, pp. 111–120.

[7] Bouneffouf, D., Laroche, R., Urvoy, T., Féraud, R., and Allesiardo, R. Contextual bandit for active learning: Active thompson sampling. In *International Conference on Neural Information Processing* (2014), Springer, pp. 405–412.

[8] Burrows, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 335–350.

[9] Chandra, T. D., Griesemer, R., and Redstone, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[11] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[12] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI* (2012), vol. 1.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *SOSP* (2007), vol. 7, pp. 205–220.

[14] Ghemawat, S., Gobioff, H., and Leung, S. The Google file system. In *Proceedings of the nineteenth Symposium on Operating Systems Principles (SOSP'03)* (Bolton Landing, NY, USA, Oct. 2003), ACM, ACM Press, pp. 29–43.

[15] Gifford, D. K. Information storage in a decentralized computer system. Tech. rep., Xerox PARC, 1982.

[16] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 15–28.

[17] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[18] Howard, H., Malkhi, D., and Spiegelman, A. Flexible Paxos: Quorum intersection revisited. *ArXiv e-prints* (Aug. 2016).

[19] Howard, H., Schwarzkopf, M., Madhavapeddy, A., and Crowcroft, J. Raft refloated: Do we have consensus? *SIGOPS Oper. Syst. Rev. 49*, 1 (Jan. 2015), 12–21.

[20] Howard, H., Schwarzkopf, M., Madhavapeddy, A., and Crowcroft, J. Raft refloated: Do we have consensus? *ACM SIGOPS Operating Systems Review 49*, 1 (2015), 12–21.

[21] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference* (2010), vol. 8, Boston, MA, USA, p. 9.

[22] Kalai, A., and Vempala, S. Efficient algorithms for online decision problems. *Journal of Computer and System Sciences 71*, 3 (2005), 291–307.

[23] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 113–126.

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[25] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[26] Lamport, L. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[27] Lamport, L., Malkhi, D., and Zhou, L. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing* (2009), ACM, pp. 312–313.

[28] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.

[29] Lamport, L. B. Generalized paxos, Apr. 13 2010. US Patent 7,698,465.

[30] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[31] Mao, Y., Junqueira, F. P., and Marzullo, K. Mencius: Building efficient replicated state machines for WANs. In *OSDI* (2008), vol. 8, pp. 369–384.

[32] Mizerany, B., Y. Q., and Xiang, L. etcd: package raft.

[33] Moraru, I., Andersen, D. G., and Kaminsky, M. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles* (2012).

[34] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 358–372.

[35] Oates, M. J., and Corne, D. Investigating evolutionary approaches to adaptive database management against various quality of service metrics. In *International Conference on Parallel Problem Solving from Nature* (1998), Springer, pp. 775–784.

[36] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 305–319.

[37] Osugi, T., Kim, D., and Scott, S. Balancing exploration and exploitation: A new algorithm for active machine learning. In *Fifth IEEE International Conference on Data Mining (ICDM'05)* (2005), IEEE, pp. 8–pp.

[38] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 288–301.

[39] Pierre Sutra, M. S. Fast genuine generalized consensus. INRIA Paris-Rocquencourt and LIP6, Universite Pierre et Marie Curie, Paris, France.

[40] Ryan, L., et al. *Google gRPC*.

[41] Schneider, F. B. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst. 2*, 2 (May 1984), 145–154.

[42] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.* (1995).

[43] Thomson, A., and Abadi, D. J. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 1–14.

[44] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 1–12.

[45] van Renesse, R., and Schneider, F. B. Chain replication for supporting high throughput and availability. In *OSDI* (2004), vol. 4, pp. 91–104.

[46] Wei, M., Tai, A., Rossbach, C. J., Abraham, I., Munshed, M., Dhawan, M., Stabile, J., Wieder, U., Fritchie, S., Swanson, S., et al. vcorfu: A cloud-scale object store on a shared log. In *NSDI* (2017), pp. 35–49.